Lattice QCD GPU programming with Quda and Grid

November 7, 2025

renwick.james.hudspith@gmail.com

Outline

- slaph interface in QUDA
  - meson functions
  - Baryon kernel

Recap on the last lecture

We learned that there is an interface for *GEMM routines in QUDA and it will translate automatically between row-major and column-major formats. The interface is shallow and it will be needed for the rest of this lecture.

There is no standard batched BLAS function call - MKL differs from cuBLAS and HipBLAS although the latter agree.

We introduced compilation and running with GRID, we talked a bit about the structure of GRID and how that translates to effective parallelism.

We saw that GRID hides the GPU interface from us but we can (and often should) drop down and use these "accelerator_for" loops, as substantial speedup is possible.

Meson currents

I showed the inner loop of the meson contractions in the last lecture, and teased it in the second. To recap we want to implement N DFTs as a small matrix batched with several right-hand-side larger matrices.

$$
N_P \left\{ \overbrace{\begin{bmatrix} e^{ip_0 \cdot x_0} & \cdots & e^{ip_0 \cdot x_{L^3}} \\ \vdots & \vdots & \vdots \\ e^{ip_{N_P-1} \cdot x_0} & \cdots & e^{ip_{N_P-1} \cdot x_{L^3}} \end{bmatrix}}^{L^3} \times L^3 \left\{ \overbrace{\begin{bmatrix} Tr(x_0, t_0) & \cdots & Tr(x_0, t_{L_t-1}) \\ \vdots & \vdots & \vdots \\ Tr(x_{L^3}, t_0) & \cdots & Tr(x_{L^3}, t_{L_t-1}) \end{bmatrix}}^{T} \right.
$$

Where the "N" is our "blockSizeMomProj" - a division of the $n1 \times n2$ noise possibilities we will run through.

What is the code doing? Well it is performing a DFT over traces of meson noise combinations $n1$ and $n2$

$$C(n1, n2, p, t) = \sum_x e^{ip \cdot x} \text{Tr}_C \left[\psi_{n2}(x, t) \bar{\psi}_{n1}(0) \right]$$

Where Tr_C is a trace over color indices.

Note that $n1$ moves slowest in this construction and t fastest. The trace is just the kernel we created in lecture 2.

As this is an interface we expect the "latticeColorVectors" to be on the host. The function we will call is:

```
void laphCurrentKernel( const int n1, const int n2,
              const int nMom,
              const int blockSizeMomProj,
              void **host_quark,
              void **host_quark_bar,
              const double _Complex *host_mom,
              QudaInvertParam inv_param,
              void *return_array,
              const int X[4])
```

All arguments here exist on the host. Here "host_mom" is a flattened array of $Np \times L^3$ Fourier twiddle factors. $X[4]$ carries the local dimensions.

Allocations

Here we allocate memory on the device, skipping creating the CPU
colorSpinorField creation

```
const size_t data_ret_bytes = nMom*X[3]*n1*n2*2*precision;
const size_t data_tmp_bytes = n_sites*2*precision*blockSizeMom
const size_t data_mom_bytes = nMom*n_spatial_sites*2*precision
void *d_ret = pool_device_malloc(data_ret_bytes);
void *d_tmp = pool_device_malloc(data_tmp_bytes);
void *d_mom = pool_device_malloc(data_mom_bytes);
```

Here we need to specify the return array size for the device "d_ret" a
temporary that holds the lattice-wide traces "d_tmp" and some space for
the copied momentum twiddles "d_mom"

---

renwick.james.hudspith@gmail.com

batched-strided ZGEMM parameters

We call the constructor

```
QudaBLASParam cublas_param_mom_sum = newQudaBLASParam();
```

And specify that the matrix to the right is transposed as our fields are in c-order

```
cublas_param_mom_sum.trans_a = QUDA_BLAS_OP_N;
cublas_param_mom_sum.trans_b = QUDA_BLAS_OP_T;
cublas_param_mom_sum.m = nMom ;
cublas_param_mom_sum.n = X[3] ;
cublas_param_mom_sum.k   = n_spatial_sites ;
cublas_param_mom_sum.lda = n_spatial_sites ;
cublas_param_mom_sum.ldb = n_spatial_sites ;
cublas_param_mom_sum.ldc = X[3] ;
```

"k" is our inner index for the matrix multiply and we will be creating an nMom $\times$ $X[3]$ output matrix.

BLAS params continued

Here we are saying that there will be no striding (i.e. we reuse the same matrix) in the left matrix of Fourier twiddles

```
cublas_param_mom_sum.a_stride = 0 ; // mom stays the same
cublas_param_mom_sum.b_stride = n_spatial_sites*X[3] ;
cublas_param_mom_sum.c_stride = X[3]*nMom ;
cublas_param_mom_sum.batch_count = blockSizeMomProj ;
```

And we will be batching the DFT over blockSizeMomProj.

```
cublas_param_mom_sum.alpha = 1.0;
cublas_param_mom_sum.beta = 0.0;
cublas_param_mom_sum.data_order = QUDA_BLAS_DATAORDER_ROW;
cublas_param_mom_sum.data_type = QUDA_BLAS_DATATYPE_Z;
```

Here we specify that the ZGEMM we want is all in row-major format

We start some copies. Here we copy all of the unbarred (inner n2) quarks to the device

```
std::vector<ColorSpinorField> quda_quark(n2) ;
for (int dil2=0; dil2<n2; dil2++) {
  quda_quark[dil2] = ColorSpinorField(cuda_quark_param) ;
  quda_quark[dil2] = quark[dil2] ;
}
qudaMemcpy(d_mom, host_mom, data_mom_bytes, qudaMemcpyHostToDe
```

As well as the exponentiated Fourier factors

The hot loop over n1 and n2 we have seen already in the previous lecture - the idea is to only load the barred quark inside the n1 loop. Do blockSizeMomProj traces into d_tmp (could and probably should be done in a MRHS way or streamed) and then call the batched BLAS DFT

Cleanup and copies back to the host

We of course must copy our object back to the host from the device

```
qudaMemcpy( return_array, d_ret, data_ret_bytes,
            qudaMemcpyDeviceToHost );
pool_device_free(d_ret);
pool_device_free(d_tmp);
pool_device_free(d_mom);
```

And free up the memory we have allocated

Note that we still aren't fully done yet as we have performed the DFT over the local dimensions only. It will be up to the end-user to finally put these correlators into global "t" indices. What we are returning is a small object so the burden on MPI shouldn't be bad.

---

renwick.james.hudspith@gmail.com

Plots from Ernest
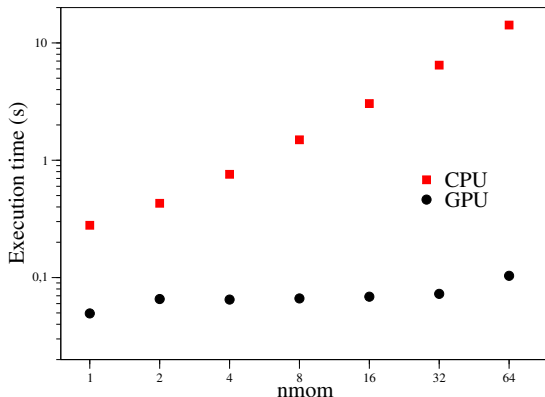
A comparison of naive single CPU vs GPU current kernels for a 16^4 local volume and $n1 = n2 = 16$ for various sizes of momenta. CPU scales linearly whereas GPU only starting to scale at large momenta.

renwick.james.hudspith@gmail.com
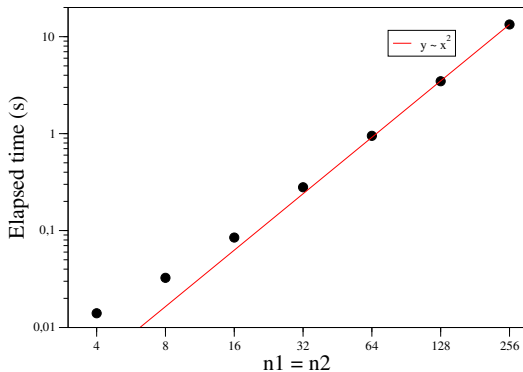
GPU scaling with $n1 = n2$ noises, should scale like $n1^2$ and does at large $n1$ where the overhead in copies and things becomes negligible (blockSizeMomProj=16, nMom=32)

renwick.james.hudspith@gmail.com

Let's look now at the baryon kernel

```
void laphBaryonKernel( const int n1, const int n2, const int n3,
          const double _Complex *host_coeffs1,
          const double _Complex *host_coeffs2,
          const double _Complex *host_coeffs3,
          const double _Complex *host_mom,
          const int nEv,
          void **host_evec,
          QudaInvertParam inv_param,
          void *return_array,
          const int blockSizeMomProj,
          const int X[4] )
```

Much like the current kernel we have noise lengths $n1, n2, n3$ we also
have 3 host_coeffs that are the explicit noise factors that will multiply
the evecs.

```
std::vector<std::complex<double>> coeffs1(n1*nEv);
std::vector<ColorSpinorField> quda_q1(n1);
for(int i=0; i<n1; i++) {
  quda_q1[i] = ColorSpinorField(cuda_q1_param) ;
  for( int j = 0 ; j < nEv ; j++ ) {
  coeffs1[j*n1+i] = (std::complex<double>)host_coeffs1[j+i*nEv
  }
}
```

And likewise for coeffs2 and coeffs3, this is just a cast into some format
we will need in a bit. Note we are creating $n1$ evecs on the device for
coeffs1 and so on for 2 and 3.

```
  {
    std::vector<ColorSpinorField> quda_evec(1) ;
    quda_evec[0] = ColorSpinorField(cuda_evec_param);
    for (int i=0; i<nEv; i++) {
      quda_evec[0] = evec[i] ;
      for( int dil1 = 0 ; dil1 < n1 ; dil1++ ) {
std::vector<std::complex<double>> cf(1,coeffs1[dil1+n1*i]) ;
blas::caxpy( cf, quda_evec, quda_q1[dil1] ) ;
      }
      for( int dil2 = 0 ; dil2 < n2 ; dil2++ ) {
std::vector<std::complex<double>> cf(1,coeffs2[dil2+n2*i]) ;
blas::caxpy( cf, quda_evec, quda_q2[dil2] ) ;
      }
      for( int dil3 = 0 ; dil3 < n3 ; dil3++ ) {
std::vector<std::complex<double>> cf(1,coeffs3[dil3+n3*i]) ;
blas::caxpy( cf, quda_evec, quda_q3[dil3] ) ;
      }
    }}
```

We create a scope where one single ColorSpinorField (device evec) will live

For every host evec we do a copy onto the device, and form the noises quda_q* by multiplying with the noise vector (after punning into a type that the device blas::caxpy wants)

We do this for every set of noises for $n1, n2, n3 < nEv$ this is quite a big memory saving and the cost of caxpys is negligible, in principle this could be MRHS-ed if it became a problem

DFT parameters

Just like the meson setup but this time we will be ordered like
$C(p, n1, n2, n3, t)$ with momenta moving slowest.

```
cublas_param_mom_sum.m = nMom;
cublas_param_mom_sum.n = X[3];
cublas_param_mom_sum.k = nSp;
cublas_param_mom_sum.lda = nSp;
cublas_param_mom_sum.ldb = nSp;
cublas_param_mom_sum.ldc = X[3]*n1*n2*n3;
cublas_param_mom_sum.a_stride = 0 ;
cublas_param_mom_sum.b_stride = nSites ;
cublas_param_mom_sum.c_stride = X[3] ;
cublas_param_mom_sum.batch_count = blockSizeMomProj;
```

Two new kernels appear!

There are two building blocks that want new QUDA kernels

colorCrossQuda(A,B,C)

does (over color indices)

$$C_a(x) = (A \times B)_a(x) = \sum_b \sum_c \epsilon_{abc} A_b(x) B_c(x)$$

And

colorContractQuda(A,B,C);

does

$$C(x) = \sum_a A_a(x) B_a(x)$$

Note that this is just like the innerProduct for the mesons but without a conjugate!

The hot zone

```
for( int dil1=0; dil1<n1; dil1++ ) {
  for( int dil2=0; dil2<n2; dil2++ ) {
    colorCrossQuda(quda_q1[dil1], quda_q2[dil2], quda_diq);
    for (int dil3=0; dil3<n3; dil3++) {
     colorContractQuda(quda_diq, quda_q3[dil3],
        (std::complex<double>*)d_tmp + nSites*nInBlock);
     nInBlock++;
     if (nInBlock == blockSizeMomProj ) {
       blas_lapack::native::stridedBatchGEMM(d_mom, d_tmp,
         (std::complex<double>*)d_ret + X[3]*blockStart,
         cublas_param_mom_sum, QUDA_CUDA_FIELD_LOCATION);
       blockStart += nInBlock ; nInBlock = 0;
      }}}}
```

Creates a "diquark" from the color cross product of q1 and q2. Color
traces with q3 and DFTs the result.

And finally the last copies and frees

```
qudaMemcpy( return_array, d_ret, data_ret_bytes,
            qudaMemcpyDeviceToHost );
pool_device_free(d_tmp);
pool_device_free(d_mom);
pool_device_free(d_ret);
```
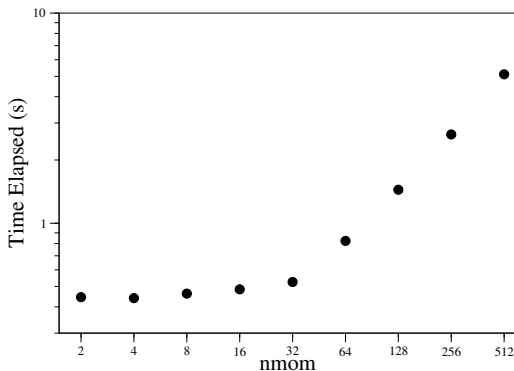
Of course the laphBaryonTripletA code works almost entirely the same way.

Homework

Can you implement the colorContractQuda and colorCrossQuda kernels?

Check against:
https://github.com/RJHudspith/quda/blob/develop/lib/color_contract.cu
https://github.com/RJHudspith/quda/blob/develop/include/color_spinor.h
renwick.james.hudspith@gmail.com

NeV $= 64$, $n1 = n2 = n3 = 16$ blockSizeMomProj$=256$, 16^4 local volume Only for many momenta do we start to scale - DFT is not where the time is spent until 64 momenta

renwick.james.hudspith@gmail.com
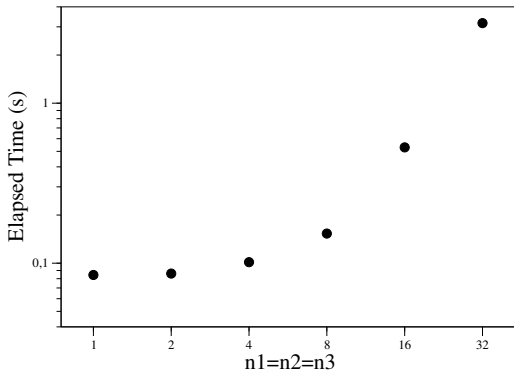
NeV = 64, blockSizeMomProj=n1*n2, 16^4 local volume still not close to asymptotic n^3 scaling before I ran out of memory
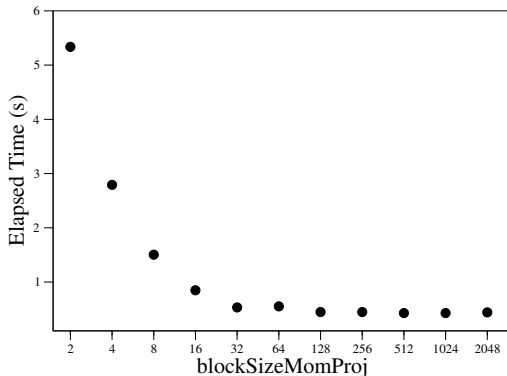
NeV = 64, nmom = 32, $n1 = n2 = n3 = 16$, 16^4 local volume A factor of $10\times$ improvement from properly tuning this parameter

renwick.james.hudspith@gmail.com

A note on memory usage for the laphBaryonKernel

d_tmp is of size (double complex)*V*blockSizeMomProj

ret is of size (double complex)*n1*n2*n3*T*nmom

q1, q2, q3 are size Nc*V*(n1/n2/n3) respectively

To summarise

Current kernel with batching is fast and dominance against the CPU can be achieved by doing a large number of momenta and a large number of noises as this is where the good scaling kicks in and the overheads are better hidden.

The baryon kernel suits this as it is doing $n1 \times n2 \times n3$ traces and DFTs. For $n* < nEv$ a significant saving in memory can be achieved by just reading evecs in when needed and using gratuitous caxpy calls.

renwick.james.hudspith@gmail.com